UCRL-ID-126480

# An MPI-IO Parallel I/O Prototype on the Cray T3D

Robert K. Yates

**February 13, 1997**

Lawrence Livermore National Laboratory

# An MPI-IO parallel I/O prototype on the Cray T3D

Robert Kim Yates
Lawrence Livermore National Laboratory

**Abstract**

We describe mpio3d, an I/O library designed to provide fast parallel transfer of data between massively parallel processors' memory and files spread over multiple storage devices. The implementation performs well on regular and irregular data distributions, and the programming interface (MPI-IO) is portable over a wide variety of computer systems. This implementation was part of an LLNL project to help develop and promote portable high-performance I/O for scientific applications.

## 1 The problem

Users of the world's most powerful computers have benefited greatly from trends toward greater parallelism, much larger memories and ever-increasing execution speeds. For example, in 1999 Lawrence Livermore National Laboratory's ASCI Blue Pacific RS/6000 SP computer from IBM will have 4096 processors with a peak processing speed of 3.2 teraflops, 2.5 terabytes of memory and 75 terabytes of disk storage. This represents an increase in computing power of two orders of magnitude over LLNL's current Meiko CS2.

Making effective use of these new capabilities is essential. But while the speed and capacity of machines like this are astonishing, the movement of vast quantities of data into and out of them has been problematic. The root of the trouble is that data are still stored on electromechanical devices such as magnetic disks whose speeds have not kept pace with processor speeds and memory sizes. To help alleviate this problem the Portable I/O Library for Massively Parallel Processors project was initiated at LLNL in 1994 and extended into 1996.

The I/O problem we address has two main aspects. The first is the technical challenge of providing fast parallel transfers of data for a wide variety of access patterns. Existing standard I/O systems work well for nonparallel sequential access of consecutive blocks of a particular size and alignment determined by the hardware/software system. To get the high aggregate throughputs needed for applications on large multiprocessors we need new mechanisms to provide concurrent access of many processes to a single file whose data are spread over multiple storage devices. Moreover, many parallel applications have complex data structures that do not match the simple well-aligned single-sized block model, so it is highly desirable to find ways to support efficient transfers of blocks of data of arbitrary size and location. The second main aspect of the problem is the need for portability: applications must run on many different computer systems. Standard programming languages such as Fortran and C and standard message passing libraries such as PVM and MPI help, but since there is currently no standard for parallel I/O, a parallel program, if it is to be portable, can do only standard sequential I/O, which is far too slow for most parallel applications' very large files. Hence an important part of this project's mission was to help develop and promote a widely-used and supported standard for parallel I/O.

## 2 Our approach

Early in the project we examined the state of the art of I/O, investigated application developers' current and projected needs, and evaluated relevant trends in computer hardware and software. This led us to identify a then-newly proposed I/O programming interface called MPI-IO [1] as the best candidate for continued in-depth research and development, based on our estimation of its technical superiority and of its potential to become widely used. Its main distinction from other proposed parallel I/O systems is that it is designed to complement the capabilities of the MPI [2] message passing interface. In particular, writing and reading files is analogous to sending and receiving messages; moreover, MPI has a very flexible "datatype" facility that is used in MPI-IO to describe the arrangement of data in processor memory and in files. Since we expected message passing, and especially MPI, to be the dominant programming paradigm for the foreseeable future (at least five years), we considered MPI-IO to be an attractive infrastructure for the development of faster and more flexible I/O techniques.

We implemented an MPI-IO prototype (which we call mpio3d) on LLNL's 256-processor Cray T3D, and ran numerous application kernels and benchmark programs to evaluate its performance and usability. This paper reports the results of the most important benchmark programs.

The performance of the new library is good. For example, a program that accesses an 800 megabyte file from 256 processors has shown write and read throughput rates up to 150 and 160 megabytes per second, respectively, when the file is distributed over 10 RAID disk storage devices. Moreover, we believe there are improvements that can be made to significantly enhance mpio3d's performance even further.

One special aspect of our research has been our attention to the needs of application programmers whose data are not evenly distributed among processors in large contiguous blocks. In irregular mesh finite element codes, for example, data are distributed in relatively small chunks of more-or-less random size and location; that is, if the data from all the processors are stored in a single file, the data from each processor are scattered into the file in randomly-sized chunks at random offsets in the file. Since transfers of data to disk are efficient only for large chunks aligned at disk block boundaries, our implementation of MPI-IO first collects small requests into large ones. The aggregation of requests is accomplished by a software cache for I/O developed by Cray Research. The effectiveness of the aggregation is shown by the fact that a program using mpio3d to randomly access a 1 GB file with data chunks distributed randomly in size between 4,000 and 524,288 bytes can still achieve a throughput of over 60 MB/sec for writes and 115 MB/sec for reads. This is good performance for such a difficult task. Moreover, MPI-IO's datatypes make handling such irregular data distributions very simple for the application programmer.

## 3 MPI-IO

As already noted, MPI-IO is a parallel I/O library that can be linked with MPI message passing application programs. The first version of MPI-IO was designed at IBM's T.J. Watson Research Laboratory in early 1994 by the creators of Vesta [3], a research prototype parallel file system. Later in 1994 MPI-IO underwent a major revision with help from researchers at NASA Ames. The revision improved the interface by incorporating MPI's derived datatypes to describe the layout of data in memory and in files. This is the version (v0.5) that is (partially) implemented by mpio3d. The group at IBM Watson Laboratory also developed a prototype MPI-IO over the PIOFS file system on the SP-2, and the NASA Ames group has built an implementation for networks and clusters of workstations using a client/server design. Despite the name, these early versions of MPI-IO were not part of the MPI standard; however, they have become the basis for the new I/O Chapter that is currently under consideration for inclusion as part of MPI-2.

We can give only a short overview of the MPI-IO user interface. The read and write operations each have twelve different forms corresponding to whether the user wants to use explicit offsets, a single file pointer shared by all processes, or a separate file pointer for each process; whether the operation should return when the transfer is initiated or wait until it is completed; and whether the operation is to be collective or independent. The latter distinction has no effect on the results except that collective operations can be faster than independent ones when large portions of the file are accessed by all the processors acting in the aggregate. Every process opens the file, each one giving an MPI datatype that specifies a pattern representing the file locations accessible to that process. Processes can then read or write the file concurrently, either collectively (i.e., each doing a single request at the same time, more or less) or independently . Of course the intent of collective requests is that the separate transfers (one from each process) may be combined into larger, more efficient requests. The read and write commands also have MPI datatype parameters that specify how the data are laid out in the process' memory .

As an example, assume an application program has distributed an irregular mesh over some set of processes, that every process has an array `local_data` of length `local_count` containing the parts of the mesh it owns (assume each part is a `real`), and an array `local_to_global_map` of indices, where `local_to_global_map[i]` gives the relative position in the file where element `local_data[i]` is to be written. First each processor sets up in its variable `local_file_type` the MPI datatype that describes where the data from that processor will be placed in the file:

```
MPI_Type_Indexed(
      local_count,
      array_of_1s,
      local_to_global_map,
      MPI_REAL,
      &local_file_type );
```

In building this type we have treated each individual real number as a separate chunk of data (hence the use of an array of ones to specify the length of each chunk). One could of course improve the treatment of sequences of data elements that are contiguous in both local memory and in the file by using a more complex type.

Next each process opens the file:

```
MPIO_Open(
      MPI_COMM_WORLD,
      “my_file_name”,
      MPIO_WRONLY,
      0,
      MPI_REAL,
      local_file_type,
      hints,
      &local_file_handle );
```

The first parameter specifies which MPI “communicator” will have concurrent access to the file. The fourth parameter is a displacement to skip over any desired number of bytes at the beginning of the file. The fifth parameter gives the “base” type, which must underlie both the file type (i.e., the sixth parameter to `MPIO_Open`) and the buffer type (i.e., the fourth parameter to `MPIO_Write`). The hints parameter can be used to provide information to the library that can be used to improve performance, e.g., striping or preallocation values.

Finally, each process executes a single MPIO_Write to transfer all its data to storage:

```
MPIO_Write(
      local_file_handle,
      0,
      local_data,
      MPI_REAL,
      local_count,
      &status );
```

## 4 Cray T3D I/O subsystem

The T3D at Livermore is a 256-processor system (128 nodes, 2 processors per node) with 64 MB of memory per processor. Each processor has a peak computational rate of 150 Mflops. The nodes are connected in a three-dimensional torus with a peak bandwidth of 300 MB/sec in each of six directions. Embedded in the torus are two I/O gateways, each rated at 200 MB/sec. Our test files were all written on /usr/tmp, which is served by a number of DA-301 disk arrays. Each array has a formatted capacity of 5.5 GB and can sustain a throughput of about 30 MB/sec.

There are two paths the data can take in going between the T3D and storage. In Phase 1 I/O, the data are routed through the host YMP. In Phase 2 the data can flow directly between the T3D and storage without passing through the host. By default, Phase 1 I/O is used. By setting an environment variable, a user can request that a job use Phase 2 I/O when the transfer requests are well-formed, i.e., when they are multiples of the size of a disk sector and are aligned at a sector boundary. In both Phases 1 and 2 the transfers are controlled by the host.

## 5 Implementation of mpio3d

Mpio3d implements only the explicit-offset, blocking, independent versions of read and write, with arbitrary file and buffer types. We tested mpio3d with the T3D-specific version of mpich, the MPI implementation developed at Argonne National Laboratory and Mississippi State University. There is a C and a FORTRAN interface to mpio3d.

For a read or a write, mpio3d reconciles the file and buffer types into a series of contiguous sections of data and calls a lower level I/O layer once for each section. The lower-level layer is called "par_io," which was developed by Cray Research employees after discussions about the need for improved handling of data blocks that could be of any size and be located anywhere in memory and in the file. The performance of mpio3d is essentially that of par_io, used well.

Par_io provides a software-implemented cache underneath mpio3d. When par_io is initiated it allocates a certain number of shared-memory "cache pages" to each processor. The size and number of cache pages can be set by the user at program start-up via a pair of environment variables. If $n$ is the size in bytes of a cache page, the first $n$ bytes of a file will be cached in processor 0, the next $n$ bytes in processor 1, etc. To move data between its buffers and disk storage, par_io uses Cray's "list I/O." List I/O extends the Unix interface to support concurrent file access to arbitrary locations.

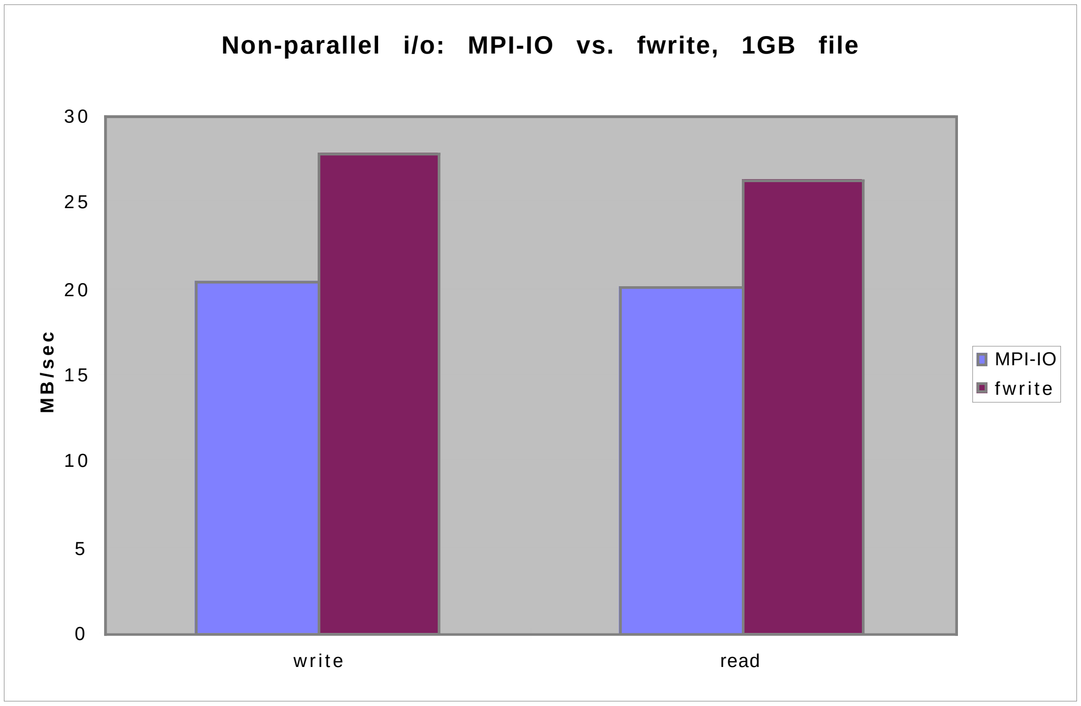
By going through the cache, the data from small, independent writes will, if they have sufficient spatial and temporal locality, be aggregated into large, cache-page-sized transfers; similarly for reads, in the opposite direction. Thus, even though mpio3d does not implement collective operations explicitly, locality often achieves the same effect.

## 6 Performance tests

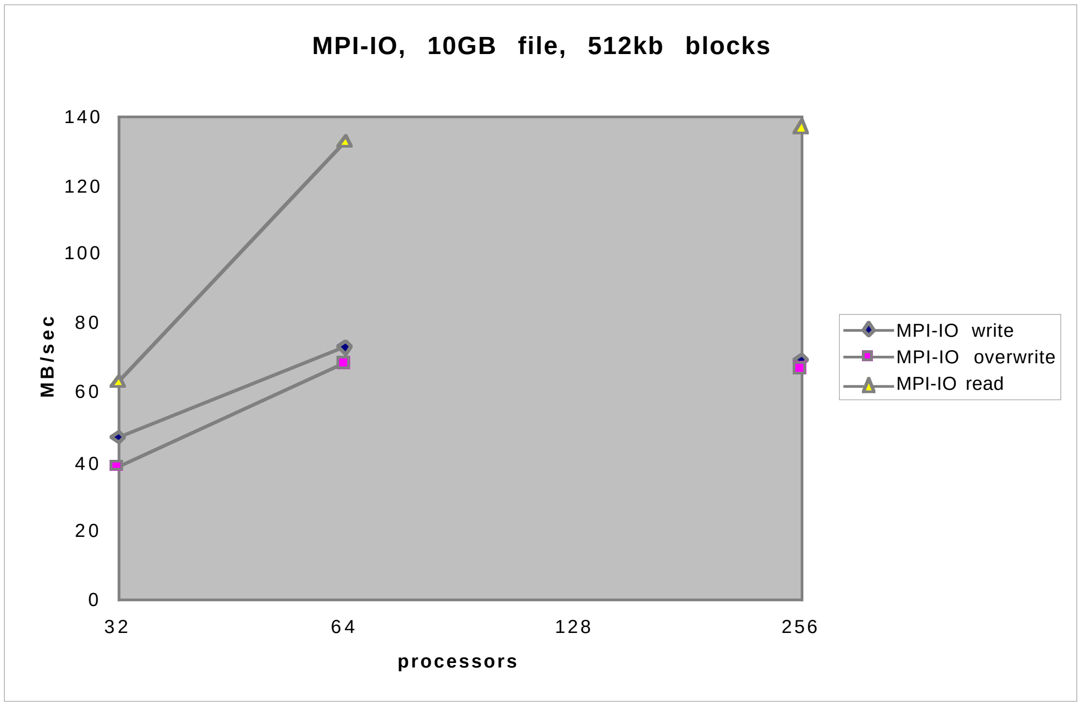
Unless otherwise noted, all files were striped over ten DA-301 disk arrays using Cray’s “user level striping” (i.e., via cbits and cblks) and accessed via Phase 1 I/O. The resulting files can be accessed by any of the standard Cray I/O methods. We set up par_io to provide two cache pages per processor, each having 3276800 bytes (this is 800 times the sector size of 4k bytes, a good choice for our striping scheme).

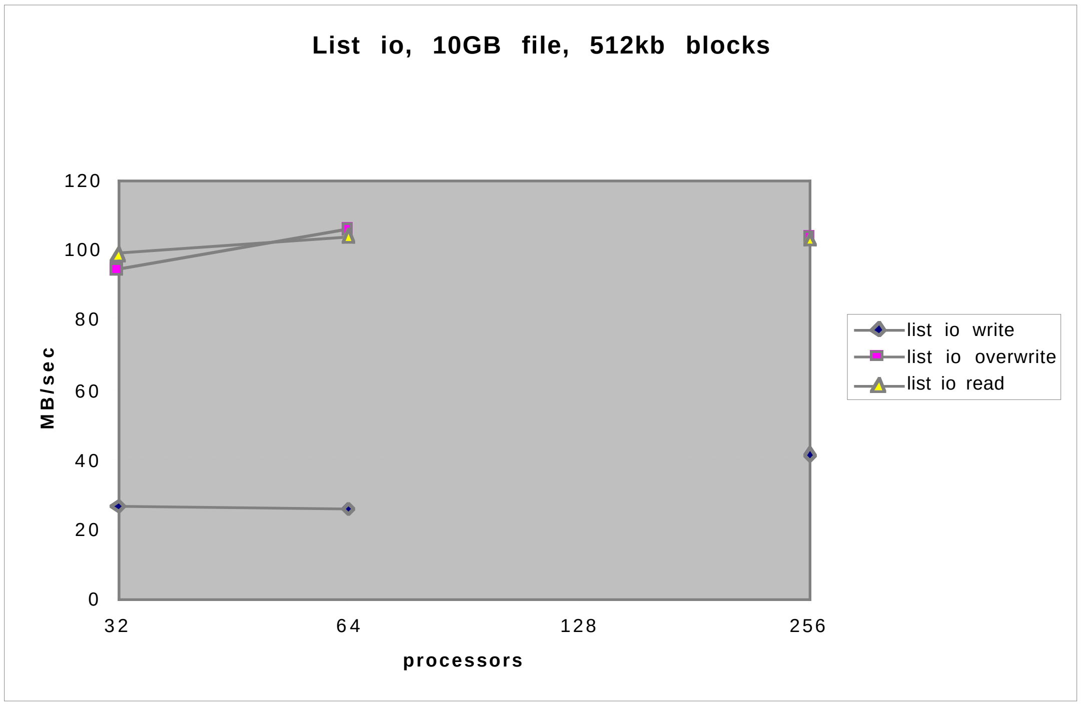
Our method of measuring execution time is very conservative. For each write or read we measure the elapsed wall clock time from before the file is opened to after the file is closed. All processors synchronize before the open and after the close, and the time we measure starts before the first synchronization and ends after the last synchronization. Because we measure wall clock time (using the routine gettimeofday) and because we compete for resources with all the other jobs in the system, the execution time of a test program can vary a lot from one run to the next. Note that even when we run with all 256 of the T3D’s processor’s, jobs running on the host YMP can also access the disk partitions we use. Unless noted otherwise, all tests were run two or three times, and the performance reported here is the best of those (i.e., the shortest time).

Our first test compares the performance of MPI-IO to the standard Unix fwrite, having one processor do a sequential write and read of a non-striped file. The overhead of going through several extra layers of control and of copying the data into and out of par_io’s cache is evident.

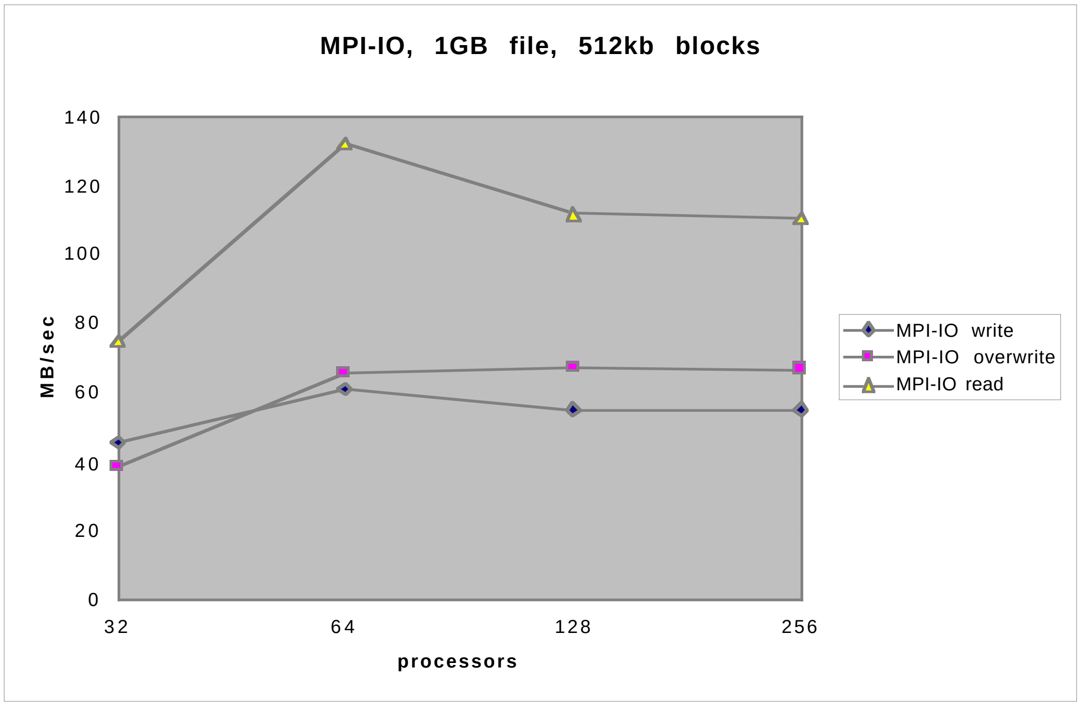


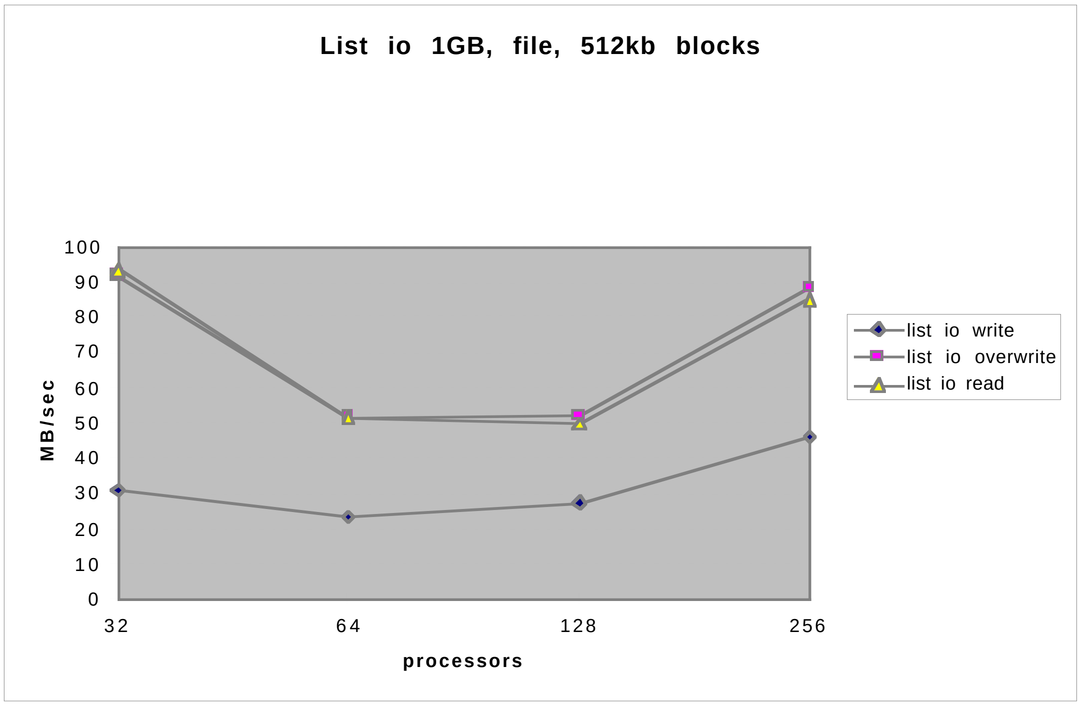
For parallel access, we compare MPI-IO to Cray's list I/O. In the next test we measured the performance of MPI-IO and list I/O reading and writing a 10 GB file, varying the number of processors. The accesses were done in units of 512kB blocks, with block 0 of the file going to/from processor 0, block 1 going to/from processor 1, etc. The "write" tests were writes to non-preexisting, non-preallocated files; "overwrite" tests were writes to preexisting 10 GB files.



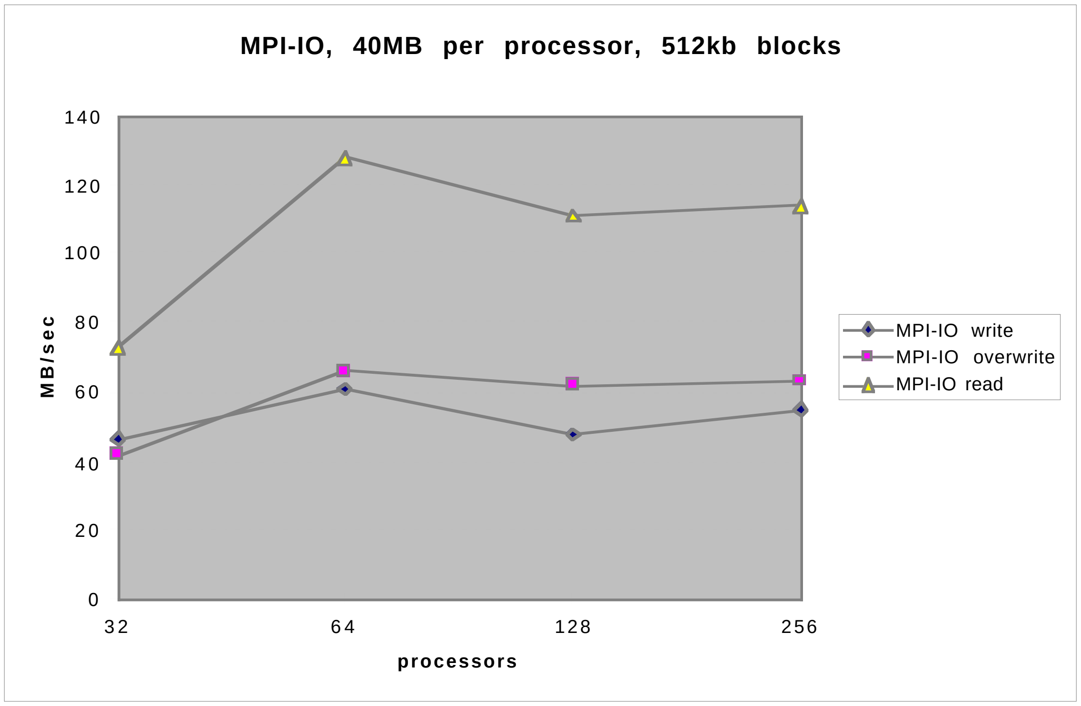


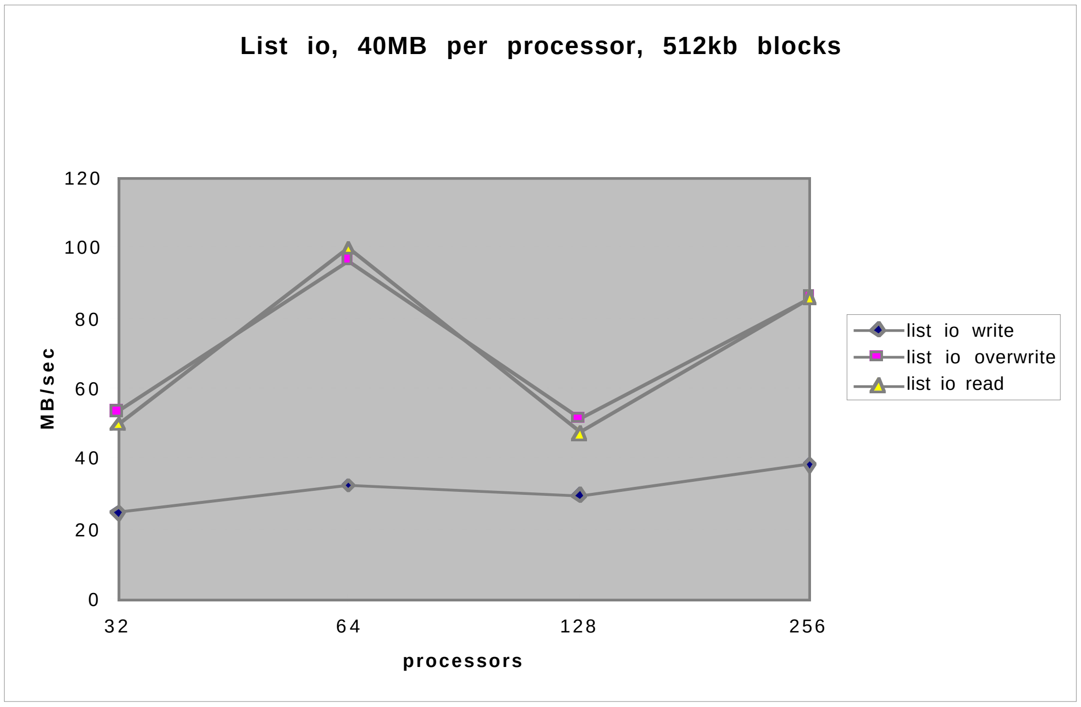
We ran the same tests on a 1 GB file:



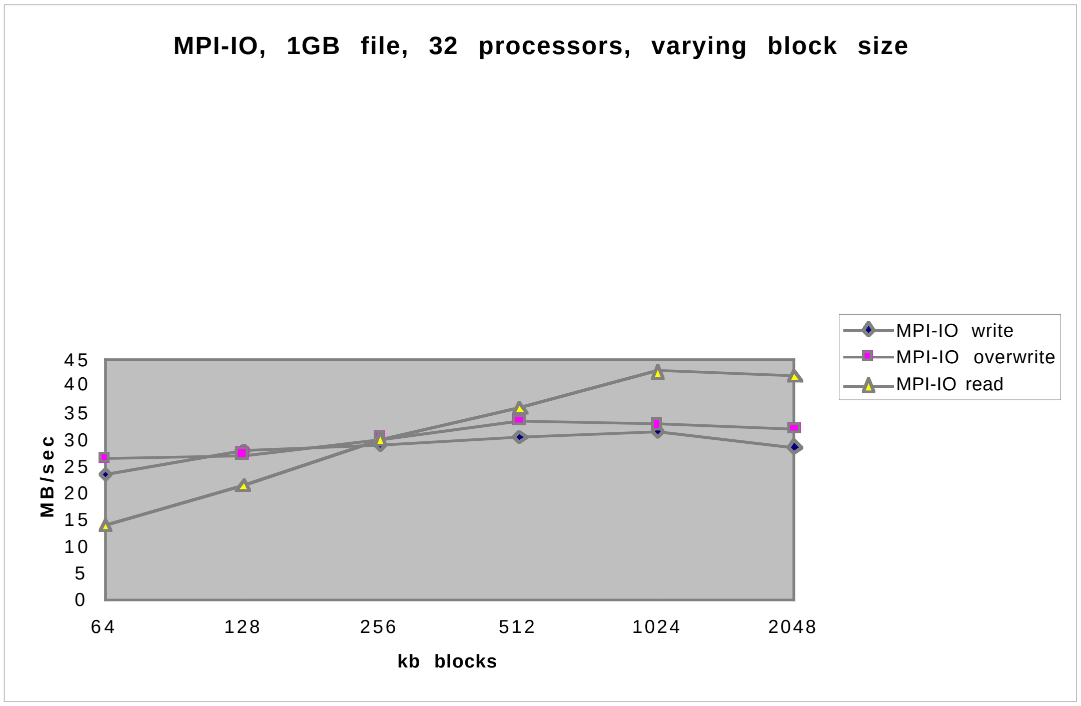


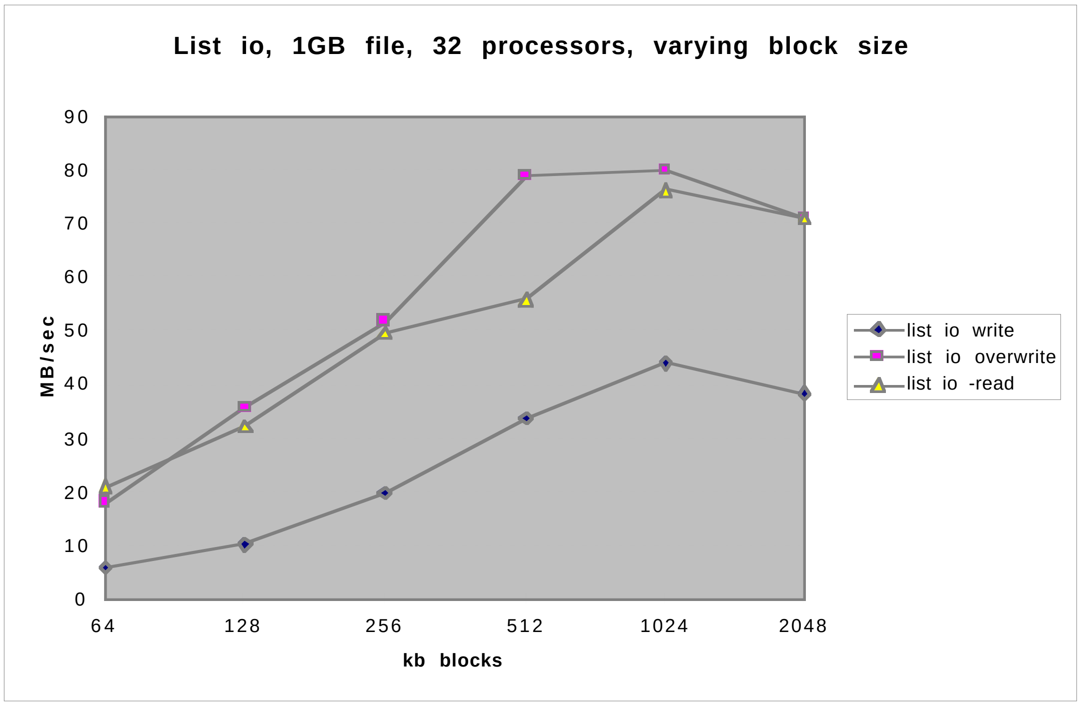
The next graphs show the results of similar tests, but keeping the ratio of the file size to the number of processors constant at 40 MB per processor.





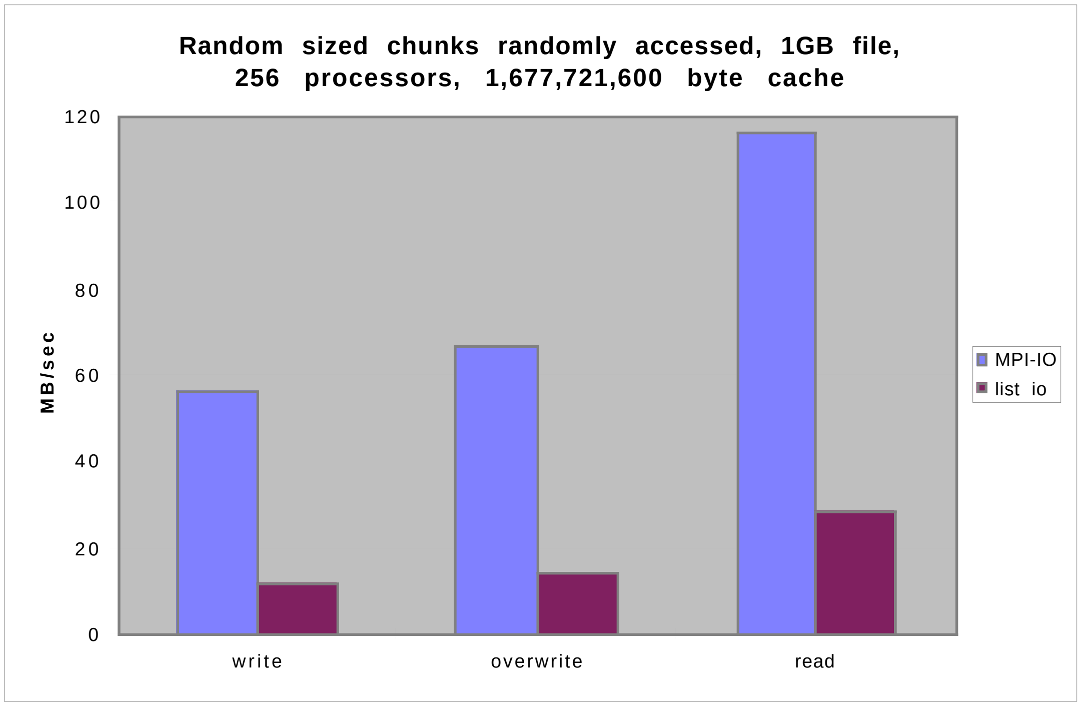
In the next test we kept the file size constant again at 1 GB and varied the block size. The number of processors is 32.

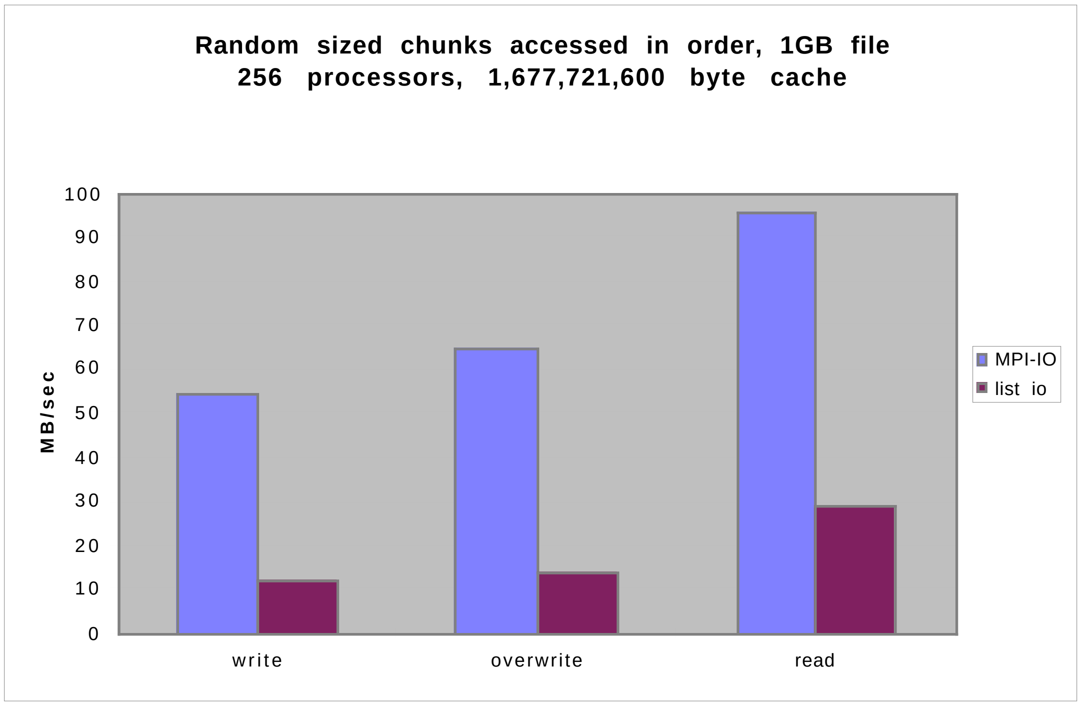




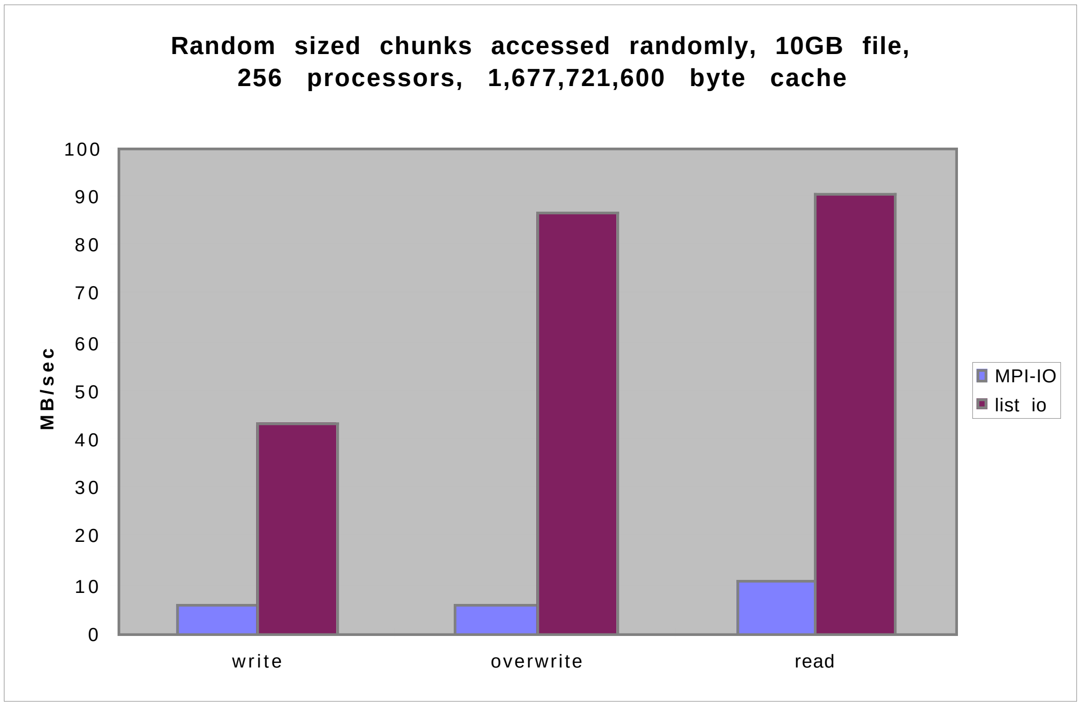
In the following tests we accessed the file randomly rather than in constant sized blocks well aligned. The chunks were randomly sized from between 4,000 to 524,288 (i.e., 512k) bytes. The chunks tile the file with no overlap. Chunk 0 is accessed by processor 0, chunk 1 by processor 1, etc., in random order in some tests, in order of ascending location within the file in others.

In the first random test each processor has a random permutation of the indices of its chunks so that it accesses those chunks in random order. Note that with 256 processors a 1 GB file fits entirely into the par_io cache (2*256*800*4096 bytes), so the performance of mpio3d is excellent in this case. In the second test each processor accesses its chunks in ascending order. Mpio3d's performance is less than in the random access test because the processors contend more for the same cache pages.

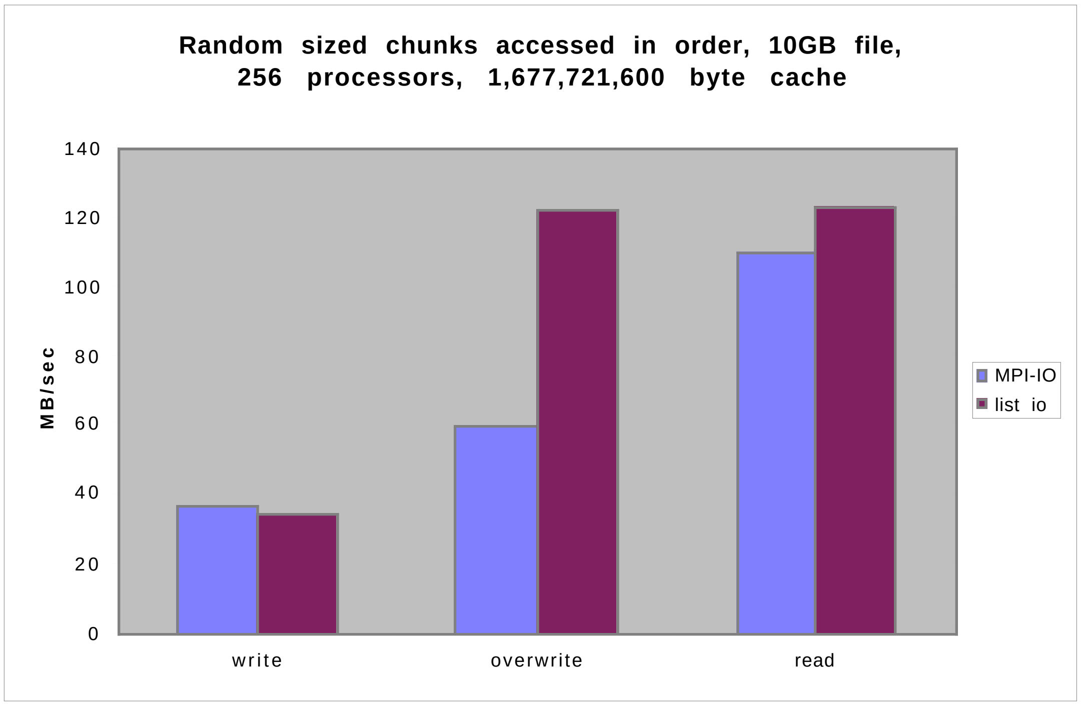




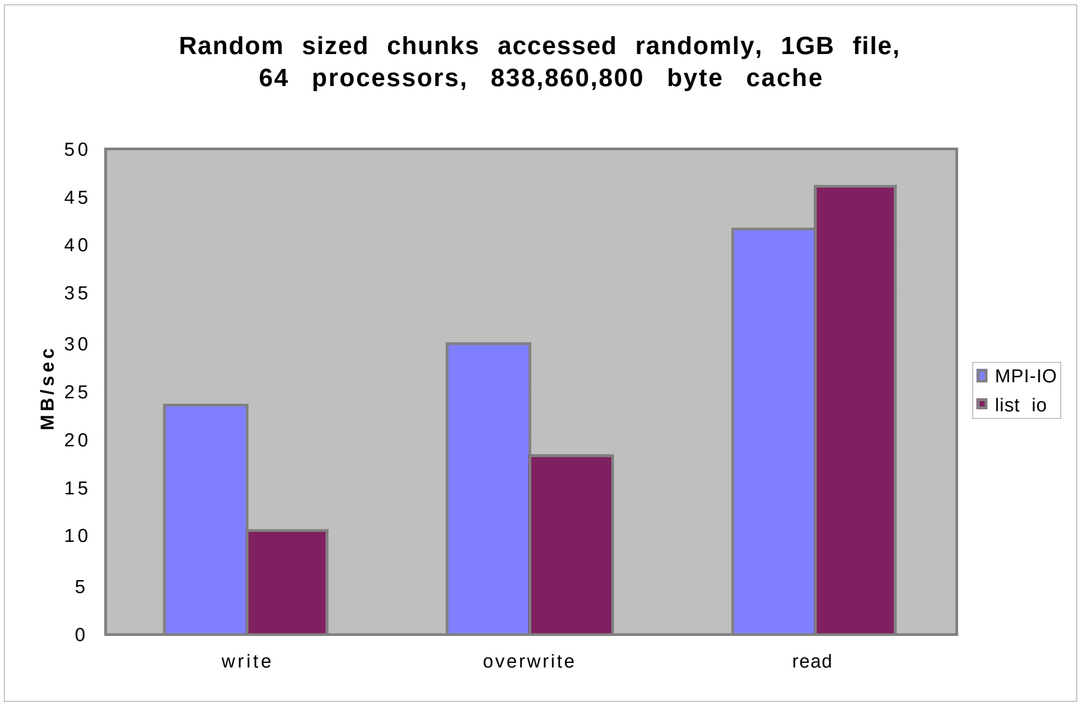
In the following six tests we access randomly-sized chunks of a file that is too big to fit into par_io's cache. In the graph below we lengthened the file to 10 GB, using 256 processors, each with 2 cache pages. In this test mpio3d performed poorly, probably due to thrashing in the par_io cache.

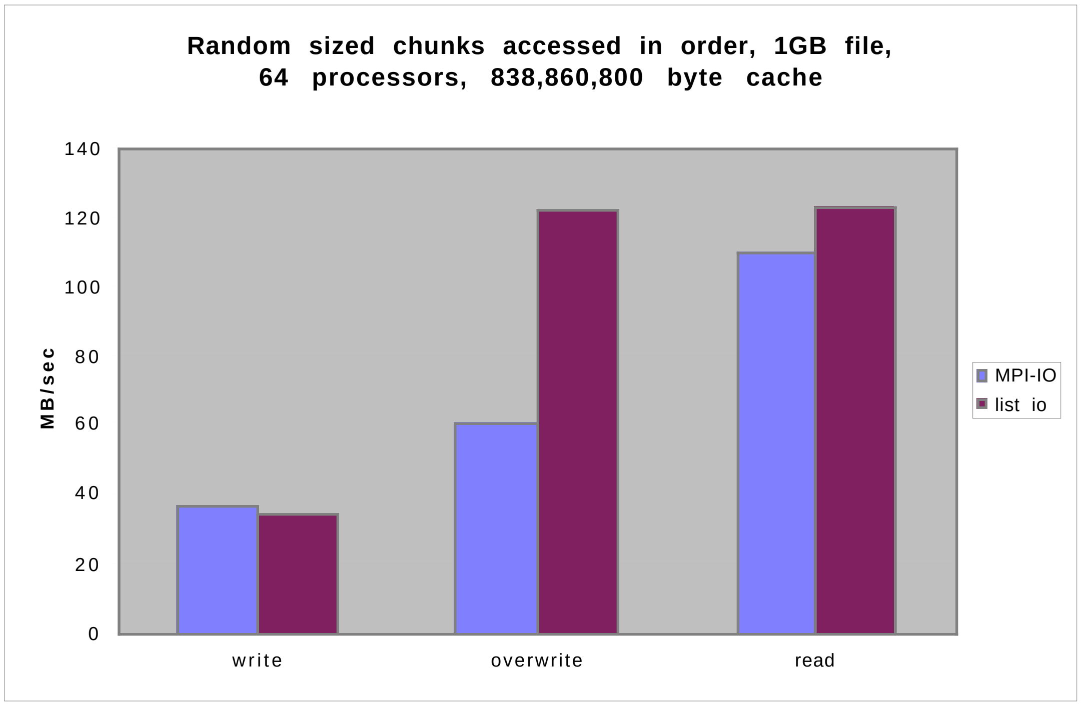


The next test is similar, but each processor accesses its chunks in ascending order of their location in the file rather than in random order. In this scenario mpio3d's caching works fairly well.

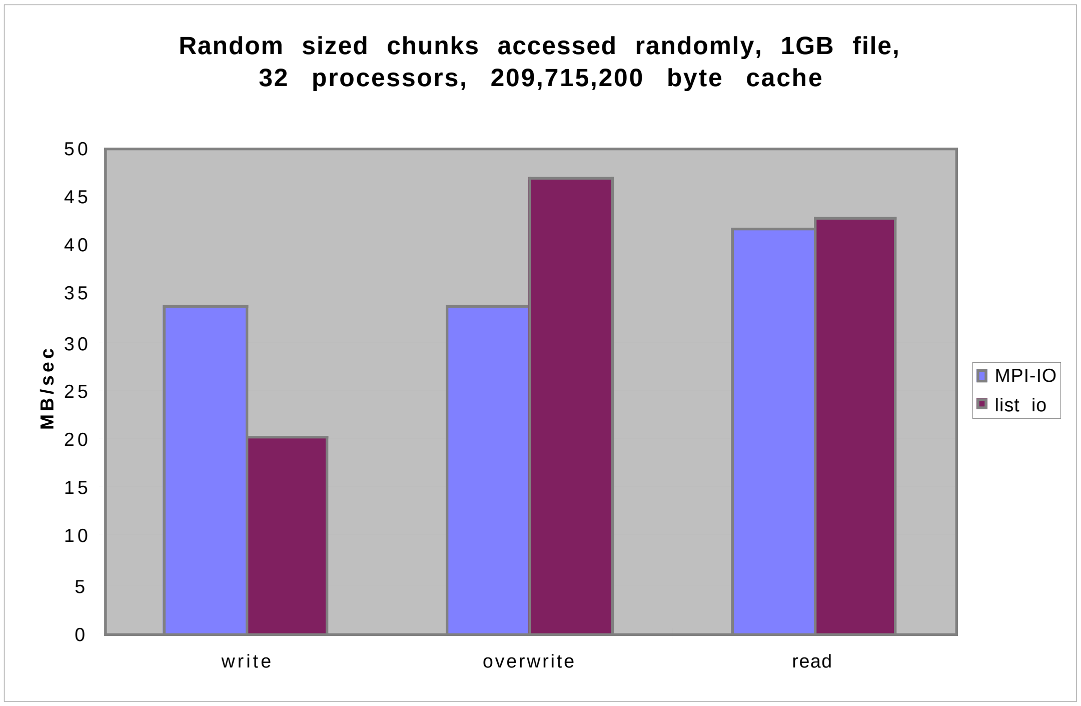


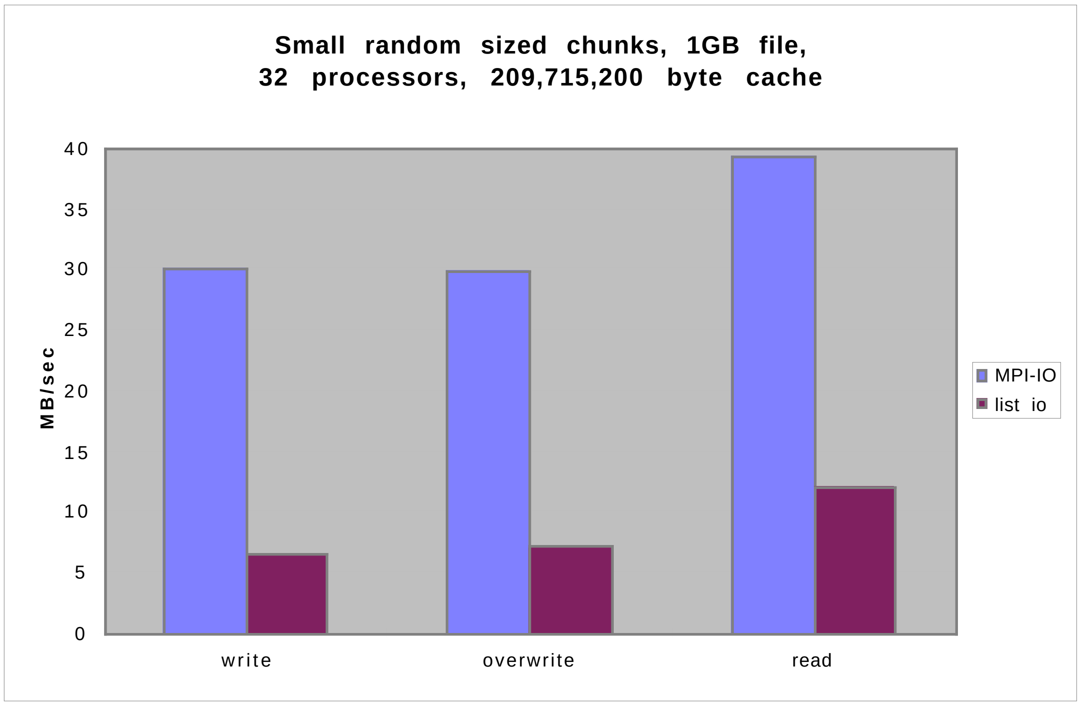
Next we increased the number of cache pages per processor to 4, reducing the number of processors to 64, accessing the blocks in random order and then in ascending order of location.





In the final two tests shown here, we reduced the number of processors to 32, with 2 cache pages each. Both tests use random access of the chunks, but the first test distributes the chunk sizes between 4,000 and 524,288 bytes as before, while the second test uses only small chunk sizes between 500 and 10,000 bytes. With small chunk sizes mpio3d performs especially well in comparison to list i/o.





## 7 Conclusion

We believe that mpio3d, the prototype implementation of MPI-IO demonstrated here, shows that MPI-IO can deliver fast concurrent access to a single file distributed over multiple storage devices, with generally good performance even for very irregular data access patterns. We have shown that a caching/aggregating layer can be especially advantageous when data blocks are not well matched to the hardware and operating system "sweet spots." Indeed, par_io's caching has since been incorporated into Cray products as "global I/O." It is likely that further study would identify ways to improve the performance of mpio3d; e.g., perhaps in some cases the cache should be bypassed altogether, and perhaps the effects of interprocessor contention for cache pages could be reduced.

It is likely that MPI-IO will become a widely used and supported de-facto standard for parallel I/O in 1997. This will enable real portability of I/O-intensive parallel programs.

By demonstrating a fast and portable I/O library, this project has made a significant contribution toward making new generations of supercomputers deliver the performance they were designed for.

## 7 Acknowledgements

We benefited greatly from the guidance of LLNL's Dick Watson for the duration of this project. Many thanks to Steve Luzmoor of SGI/Cray Research, the principal developer of par_io. Thanks also to SGI/Cray's Scott Emery and LLNL's Greg Tomaschke for their help and cooperation in running the many test programs on LLNL's T3D.

## References

[1] Peter F. Corbett, Dror G. Feitelson, Yarsun Hsu, Jean-Pierre Prost, Marc Snir, Sam Fineberg, Bill Nitzberg, Bernard Traversat, and Parkson Wong. *MPI-IO: A parallel file interface for MPI.* Technical report, 1994. Available at http://lovelace.nas.nasa.gov/MPI-IO/mpi-io.html.

[2] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker and Jack Dongarra. *MPI-IO: The Complete Reference.* MIT Press, 1995. Lots of information about MPI is available at http://www.mcs.anl.gov/mpi/.

[3] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, Sandra Johnson Baylor. *Parallel access to files in the Vesta file system.* Proc. of Supercomputing 93, IEEE Computer Society Press, 1993.